NASA-TM-111485

Dynamic Optimization

PHILIP LAIRD
AI RESEARCH BRANCH, MAIL STOP 269-2
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94025, USA

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-92-09

April, 1992

To appear in: Machine Learning International Conference, July, 1992

Dynamic Optimization

Philip Laird[*]
AI Research Branch
M.S. 269-2
NASA Ames Research Center
Moffett Field, California 94035, U.S.A.

Abstract

We distinguish *static* and *dynamic* optimization of programs: whereas static optimization modifies a program before runtime and is based only its syntactical structure, dynamic optimization is based on the statistical properties of the input source and examples of program execution. Explanation-based generalization is a commonly used dynamic optimization method, but its effectiveness as a speedup-learning method is limited, in part because it fails to separate the learning process from the program transformation process. This paper describes a dynamic optimization technique called a *learn-optimize cycle* that first uses a learning element to uncover predictable patterns in the program execution and then uses an optimization algorithm to map these patterns into beneficial transformations. The technique has been used successfully for dynamic optimization of pure Prolog.

1 Introduction

Program "optimization" is the task of replacing a program (or planner, theorem-prover, etc.) by a semantically equivalent one with superior performance. "Semantic equivalence" means that both the original and the optimized programs compute the same input/output relation. Let us differentiate between two kinds of optimization: *Static optimization* methods apply before program execution; following analysis of the local and global structure, syntactic forms are replaced by equivalents that are expected to perform as well or better, regardless of the input problem. Examples include the familiar code-optimization methods for compilers. *Dynamic optimization*, by contrast, uses experience gained from the actual execution of the program to improve its expected performance on subsequent runs. Memoization, explanation-based generalization, and unfold/fold transformations are familiar methods of dynamic optimization.

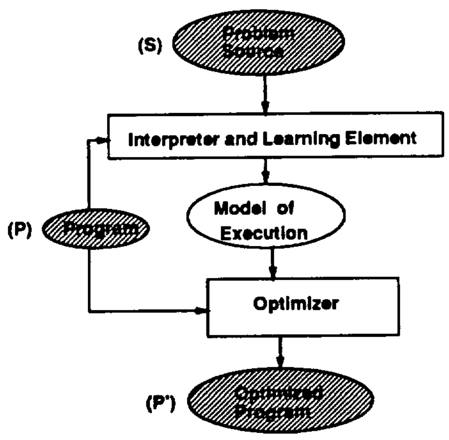

Figure 1: Dynamic Optimization Method

Currently most dynamic optimization methods are based on a "caching" model, whereby individual problem solutions (possibly after being generalized somewhat) are either remembered or discarded. A difficulty with caching techniques is that the learned information may sometimes lead to program transformations that ultimately degrade, rather than enhance, program performance—particularly with highly recursive programs. Various methods of utility analysis, e.g., (Gratch and Dejong, 1991; Markovitch and Scott, 1989; Minton, 1989; Shavlik, 1990), have been proposed to address such problems as "expensive chunks" and "generalization-to-N". The fundamental drawback, however, is much deeper: *by intertwining the learning and the transformation processes, the learning process may never converge, and as a result, finding truly effective transformations can be difficult or impossible.*

[*]Email: laird@pluto.arc.nasa.gov

The primary contribution of this paper is a new method—called the *learn-optimize cycle*—for separating the learning task from the optimization task. We consider a model (Figure 1) in which both a source program **P** and a source **S** of problems are the input; the task is to construct an equivalent program **P**$'$ whose expected performance is as good as that of **P**, and hopefully much better, on problems drawn from **S**. A learning element is used to observe a number of program executions. Naturally, the sample size must be large enough to detect statistically regular features in the examples. Then a separate algorithm (the "optimizer") is used to construct a new program that, with high probability, performs as well or better on **S**. A series of such learning and optimizing passes are used to find a (locally) optimal program.

To test the method, I built a prototype of a dynamic program optimizer for Prolog programs. I first modified a Prolog compiler so that compiled programs will pass information about their execution to a learning element called a *TDAG*. I then compiled the target program and collected data from a set of its executions, drawing problems from a particular randomized problem generator. After the learning element had stabilized, I used the learned data to modify the program using two transformation techniques: clause reordering and unfolding. Both techniques are known to preserve the program semantics (Sterling and Shapiro, 1986; Tamaki and Sato, 1984), so the optimized program was equivalent to the original. This learning/optimization process was then repeated until the optimizer could find no more transformations. The performance of the resulting program was then compared to that of the original. Experimental results are summarized later.

In this paper we first review related work and then describe briefly the learning element (TDAG) and its application to learning Prolog execution properties. We then show how the optimizer uses the TDAG information to modify the source program. Experimental procedures and results are given, followed by a critique of this methods with ideas for futher development.

2 Related Work

The objective of this research is to find a practical way to do speedup learning that rests more solidly on first principles than I have found in machine-learning papers. Recent papers, such as (Etzioni, 1990; Laird, 1991; Laird and Gamble, 1990a; Letovsky, 1990; Subramanian and Feldman, 1990), have shown how important the distribution and the order of the examples is in EBG-based systems and how it affects the estimates of utility of learned rules. Etzioni, for example, noticed that on some domains Prodigy/EBL (Minton, 1989) was learning rules that could be found just as effectively, and more efficiently, using a static learner.

Subramanian and Feldman (Subramanian and Feldman, 1990) demonstrated that one could feasibly predict the utility of certain transformations, and that unfoldings of only a few levels, instead of the EBG method of unfolding the entire solution, were worthy of study. The idea of incorporating costs and probabilities into the TDAG projections was inspired by a paper by Yamada and Tsuji (Yamada and Tsuji, 1990; Yamada, 1992), whose analysis showed that online statistics could be used to avoid utility problems without the need to benchmark each change individually on a set of example problems as in the Prodigy system. In recent work (Segre *et al.*, 1992) a combination of optimization methods, including caches and dynamic reordering, have been applied to the task of improving the performance of an automated deduction system, with considerable success.

A number of investigations of how to find an optimal ordering of conjunctive queries have pointed the way for several researchers to base reorderings on costs and probabilities. Smith and Genesereth (1985) is a good example, and Greiner (Greiner, 1989) builds on these results with some of his EBL-optimization work. Working with Orponen (Greiner and Orponen, 1991), he has developed this idea into a dynamic optimization algorithm for query databases—one that is truly "optimal" (in the sense of PAC learning), in contrast to this and other work where "optimize" is a misnomer for "improve." To date, however, their analysis applies to a restricted database, and no implementation of the algorithm has been reported.

The idea of using learning to devise program speedups and then to incorporate them into the original source program was stolen from PROLEARN (Prieditis and Mostow, 1987), one of the first dynamic program optimizers and possibly the first to employ partial evaluation techniques effectively. This approach stands in contrast to the practice of learning "search-control" rules *per se* (as in SOAR and Prodigy), without trying to convert them into program transformations

Gooley and Wah (Gooley and Wah, 1989) investigated extensively the use of Markov models of execution in order to speedup Prolog programs. In their transformations they reordered both the clauses and the subgoals within a clause, using costs and probabilities to determine the best ordering. Furthermore, their aim was to support optimization of true Prolog, including cuts and second-order constructs. Learning was not the focus of their work, nor did they attempt to choose different orderings at different places in the proof or to perform unfoldings, but their encouraging results provided a number of good ideas. Indeed, this work can be viewed as extending theirs to include clause unfolding and admit reorderings based on context.

3 The TDAG

Our optimization method begins with a learning phase. Almost any algorithm that learns to predict sequences could be used; we developed the TDAG algorithm, however, expressly for this problem. An efficient way to learn to predict sequences of discrete symbols, it has other applications and is discussed in detail elsewhere (Laird, 1992). Here we shall give only the main ideas as they apply to Prolog optimization.

Consider an input source that generates a continual stream of symbols (e.g., "q x b q s s a v s..."), and suppose we want to learn to predict the next symbol probabilistically (e.g., the next symbol will be q with probability 0.7 or x with probability 0.24, etc.). A possible approach is to model the input source as a Markov model; unfortunately no tractable method for learning a general family of Markov models is known (Abe and Warmuth, 1990; Laird, 1988). The TDAG approach, which draws on ideas from adaptive data compression (Bell *et al.*, 1990), offers a practical compromise that ameliorates many of the theoretical problems associated with Markov models.

We proceed as follows. Let a_i be the i'th symbol to arrive in the input stream. We define the set of *suffixes* (at time i) to be the set of i strings, $a_1a_2\ldots a_i$, $a_2\ldots a_i$, $\ldots$, $a_{i-1}a_i$, and a_i. We can keep a table in which we count the occurrences of each suffix of the input as each input symbol is observed. For example, after observing the four input symbols "a b b b" we would have a table containing a (1 occurrence), b (3 occurrences), ab (1), bb (2), abb (1), bbb (1), and abbb (1).

Of course, the size of a table of suffixes grows quite rapidly—potentially as the square of the number of input symbols—so maintaining such a table is not practical. But such a table can be used to predict the probability of the next symbol, as follows. Suppose that we have seen 100 input symbols so far, and that the past three input characters were "...a b b". Assume the suffix abb appears in the table with a count of 30 times and that the suffix abba occurs 6 times. This means that six times out of thirty, the sequence abb has been followed directly by the character a; hence we can estimate the likelihood of next seeing an a as 6/30 if we base our prediction for the next character on the preceding three characters. We could also base the prediction on the preceding n characters, for any n from 0 to 99.

Besides predicting the next symbol, we can also compute a *prior probability* for each suffix in the table. The prior probability of a suffix S of length k is the probability that the next k input symbols will be S, given no information about the preceding characters. In our example, abb has an estimated probability of 30/97, since it has occurred 30 times out of a total of $100 - 3 = 97$ suffixes of length 3.

This algorithm will remain impractical unless we limit the growth of the suffix table. Let us do so by removing all suffixes $S \cdot x$ (where S is a string and x a symbol) for which the prior probability of S is less than some value θ. The parameter θ is a positive fraction chosen by the user on the basis of the amount of available storage. In the preceding example, if $\theta < 30/97$, the table entry abba will be kept; otherwise it will be discarded. In typical cases the size of the table will be limited to about $O(A\theta^{-1})$ entries, where A is the number of distinct input symbols, although periodicities in the input source can still cause the size of the table to grow without limit. We must, therefore, also impose an upper limit D to the length of any suffix stored in the table—a limit that will rarely be reached in practice.

This also suggests a reasonable way to decide which suffix to use to predict the the next character: use the longest suffix S whose count is at least M and for which there is at least one extension $S \cdot x$ in the table, where x is a symbol. The parameter M is chosen based on the desired confidence in the prediction probability. Other ways to formulate predictions are possible, but this one is effective, simple, fast, and principled.

We now have an algorithm that is nearly practical. The only additional improvement is to structure the table as a tree for efficiency. The root of this tree is labeled by the empty string; and if the suffix S and its one-character extension $S \cdot x$ (where S is a string and x a character) are both in the table, then in the corresponding tree there is an edge from the node labeled S to node labeled $S \cdot x$. For example, if abb and abba are entries in the table, then the node labeled abb will have among its children a node labeled abba; if there are no extensions of abb in the table, then abb occurs as a leaf in the tree.

Luckily there are simple, efficient algorithms (not given here) for updating the tree as each new input symbol arrives and for predicting the next symbol probabilistically. We call the resulting tree a *TDAG* and refer to the algorithm as the TDAG learning element. It can be shown that the algorithm converges in the limit to a useful approximation of any Markovian input source.

For dynamic optimization we shall need to learn from Prolog proofs, which are trees, not strings. It is easy to extend the TDAG to learn a class of tree-structured sequences called *multi-strings*. Here each input symbol x comes with a unique integer ≥ 0 called its *multiplicity* ν, which we indicate by writing x_ν. Formally, a *multi-string* consists of a symbol x_ν concatenated with ν multi-strings. For example, "$\mathtt{a}_2$ $\mathtt{c}_0$ $\mathtt{b}_1$ $\mathtt{c}_0$" denotes a multi-string in which the symbol a has multiplicity two and thus is followed by two multi-strings: one consisting of only the symbol c, and one consisting of the

multi-string "b$_1$ c$_0$". Multi-strings are most easily exhibited as ordered trees in which the root node x_ν has as children ν subtrees representing the ν multi-strings that follow x. Note that ordinary strings are just a special case of multi-strings in which each symbol has multiplicity one except the last, which has multiplicity zero. Generalizing the TDAG to learn multi-strings is easy: just as a string TDAG makes a prediction of which symbols are most likely to follow the recent input symbol x_1, a multi-string TDAG makes ν predictions, one for each of the successors for the most recent input symbol x_ν. Converting a TDAG algorithm for strings into one for multi-strings is a simple matter of replacing some single-valued elements into arrays of size ν and using a stack to keep track of our depth in the multi-string.

4 Using a TDAG to learn Clause Sequences

Logic programs represent search problems in which the task is to find a clause $[\mathcal{C}]$: $H \leftarrow T_1, T_2, \ldots$ whose head H unifies with the input goal and whose subgoals T_i (after applying a unifying substitution) are all refutable. If we can predict which clause should be chosen for any given goal, then the cost of running the program is linear in the size of the solution. Our intention is to use a TDAG to guide us to the right clauses during the proof. Also, unfolding part of a proof reduces the size of some solutions and potentially changes the search order. We want to use a TDAG to tell us which unfoldings will improve the average cost of solutions, not just the cost of a single solution. Other program transformations are possible; we limited our research to these two since they preserve the semantics of the program, are frequently performed, and are relatively easy to understand.

Refuting a goal G results in a proof tree (Sterling and Shapiro, 1986) whose root is the goal G and whose children are proof trees for each subgoal generated by a resolution step. Given the proof tree one can easily derive a *clause-name tree*, in which each node of the proof tree is re-labeled with the name C of the clause used to resolve the goal or subgoal. For example, in Figure 2, we show such a clause-name tree for the three-step proof of the goal $G = p(f(a))$ using a program which will serve as a running example throughout this paper.

The key observation is that a clause-name tree is a multi-string; therefore sequences of clause-name trees can be learned using a TDAG. Each clause C has a fixed number ν of terms in its tail; thus each occurrence of C in the clause-name tree has ν subtrees whose root nodes are labeled by the names of the clauses used to resolve the subgoals. Thus the number ν of antecedents in the body of the clause C is its multiplicity.

The basic idea is that, by learning from a sequence of clause-name trees, we simultaneously learn to predict which clauses will succeed at different points in the proof. In order to improve program *performance*, however, both the *likelihood* of success and the *expected cost* of the effort need to be estimated. Consequently we shall gather cost information as well as likelihoods in our TDAG.

The TDAG learning element is used as follows. First, the target program is changed to an equivalent program in which each clause $[\mathcal{C}]$: $H \leftarrow T_1, T_2, \ldots$ is replaced by a pair of clauses: $[\mathcal{C}_1]$: $H \leftarrow \text{Tail-}T_1, \text{Tail-}T_2, \ldots$. and $[\mathcal{C}_2]$: $\text{Tail-}H \leftarrow \text{Tail-}T_1, \text{Tail-}T_2, \ldots$.. For example, the program in Fig. 2 is transformed as shown in Fig. 3. This transformation helps to distinguish clauses used to resolve the main goal from those used to resolve subgoals and provides more context within the execution on which to condition the code transformations.

For each input problem, the Prolog interpreter solves the problem while building a clause-name tree.[1] Whenever a clause C is used to try to refute a goal, a measurement is made of the cost $\$C$ of applying that clause (say, by measuring CPU time or counting unifications) and refuting its subgoals. If the clause fails, the name of the failing clause and the cost of attempting it are stored as data with the tree. If it succeeds, the name of the successful clause and the cost of finding the solution are stored in the node, and its child nodes are recursively constructed from the results of resolving its subgoals. Note that both success and failure costs are accrued.

Next, the tree is passed to the multi-string TDAG algorithm, one node at a time, in pre-order. In addition to storing the clause-names as symbols and counting their successors, we also count the total number of attempts (successful or otherwise) to use that clause and the total cost of all such attempts. The TDAG, therefore, contains enough information to predict the probability that each clause will successfully resolve a given subgoal and the expected cost of applying the clause.

As more input problems are solved and the resulting clause-name tree statistics are passed to the TDAG, the accuracy of the information increases. Unfortunately without strong assumptions about the problem source, there is no theoretically justified way to compute the number of input problems needed to guarantee that the TDAG will achieve a given level of accuracy. The practical method I used was to feed the TDAG some number m of problem results and com-

[1]For our implementation second-order program elements such as negation-by-failure and `call` were allowed, but these structures appeared as leaf nodes in the clause-name tree, without any analysis of their proof structure. Non-logical constructs like cuts were not allowed.

```
[CP1]: p(a).
[CP2]: p(f(X)) <- q(h(X)),p(X).
[CQ1]: q(h(X)).
[CQ2]: q(b).
```

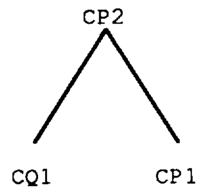

Figure 2: A simple clause-name tree. The program is shown on the left with clause labels in square brackets. To the right is the clause-name tree for the proof of the goal $p(f(a))$.

```
[C1]: p(a).
[C2]: p(f(X)) <- tail-q(h(X)),tail-p(X).
[C3]: tail-p(a).
[C4]: tail-p(f(X)) <- tail-q(h(X)),tail-p(X)).
[C5]: q(h(Y)).
[C6]: q(b).
[C7]: tail-q(h(Y)).
[C8]: tail-q(b).
```

Figure 3: Initial transformation of the program in Figure 2.

pare the results to those of a TDAG built from $3m/2$ problem results; if the prediction probabilities differed significantly, I increased the sample size. The largest number of problems I needed for learning was 300, so convergence is reasonably fast.

Summarizing, each node of the TDAG tree contains the name C of a clause, the number of attempts to satisfy a goal using that clause, the number of successful attempts, the total cost of those attempts, and the usual TDAG likelihoods for each of its subgoals. In Fig. 4 we show the structure of a possible TDAG resulting from executions of the program in Fig. 3, assuming that "p" is the predicate of the main goal. The root node has two successors $C1$ and $C2$, the clause names for predicate p. $C1$ is a leaf because clause $C1$ has no antecedents. $C2$ has two subtrees, one for each of its two antecedents. The "`tail-q`" subtree has two clauses ($C7$ and $C8$) as children. The probability $p(C7)$ (not shown) estimates the likelihood that clause $C7$ will successfully resolve the first subgoal. The cost $\$C7$ estimates the expected cost of using $C7$ to refute the first subgoal of $C2$ in this context. (Similarly for $C8$.)

The other subtree of $C2$ has two children, $C3$ and $C4$, whose statistics apply to the second antecedent ("`tail-p`") of $C2$. Clause $C4$ is expected to have two further subtrees below it, corresponding to the two antecedents in that clause.

5 The Optimizer Algorithm

In this implementation the optimizer has available two program transformations:

- Clause reordering: Change the order in which the clauses for a predicate p are attempted. For example, to solve the `tail-q` subgoal of clause $C2$ in Fig. 3, clause $C7$ will be tried before $C8$ by virtue of its position in the program. To reverse this ordering—but without affecting other calls to `tail-q`—we first change clause $C2$ as follows: `[C2']: p(f(X)) <- g218(h(X)),tail-p(X).` (`g218` is a new predicate symbol) and add these new clauses:

  ```
  [C9]:  g218(b).
  [C10]: g218(h(y)).
  ```

  Clauses $C7$ and $C8$ are unaffected. The new predicate `g218` has the same semantics as `tail-q` except for the order of its clauses.

- Unfolding: Resolve an antecedent of one clause with the head of another, resulting in a new clause. For example, if clause $C3$ is the most likely choice for solving the `tail-p` subgoal in clause $C2$, we can replace $C2$ by the following two clauses:

  ```
  [C2.1]:  p(f(a)) <- tail-q(h(a)).
  [C2.2]:  p(f(X)) <- tail-q(h(X)),g777(X).
  ```

  and add the clause:

  ```
  [C11]:  g777(f(X)) <- tail-q(h(X)),
                        tail-p(X).
  ```

  Clause $C1$ remains unchanged. The new procedure `g777` is derived from `tail-p` but omits the clause already unfolded into $C2$.

Decisions about which transformations to apply and where are based on the TDAG data collected during the learning phase. Referring to Fig. 4, suppose that in the TDAG there is a $C2$ node whose first subgoal ("`tail-q`") has clauses $C7$ and $C8$ as children, with estimates for $p(C7)$, $\$C7$, $p(C8)$, and $\$C8$, resp. By a well-known result, the optimum ordering

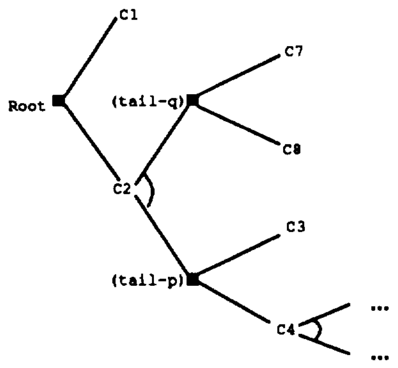

Figure 4: Sample TDAG structure.

for these two clauses is in decreasing order of the quantity $p(C_i)/\$C_i$. Thus we can quickly find applicable clause reorderings from the information in the TDAG. Note that, since clause $C8$ is attempted only on instances where clause $C7$ fails, the estimate for $p(C8)$ is actually an estimate for $p(C8 \mid \neg C7)$; hence the true optimum ordering may not always be chosen.

The analysis of unfolding transformations is more difficult, and most easily described by example. The utility of unfolding come partly from the economy of combining steps and partly from changing the order in which subgoals are resolved with their clauses. In Fig. 3, for example, when clause $C2$ is invoked and its two subgoals are resolved, the clauses are tried in the following order: $(C7, C3)$, $(C7, C4)$, $(C8, C3)$, $(C8, C4)$. But if clause $C3$ is unfolded into the second subgoal as in the above example, this order becomes: $(C7, C3)$, $(C8, C3)$, $(C7, C4)$, $(C8, C4)$. The risk is that unification costs will increase when all subgoals fail since there is an additional clause in the procedure. Whether the unfolding will improve the expected cost of the program is important to predict with high confidence, since (unlike clause reorderings) unfoldings cannot be undone later by our optimizer.

Consider the unfolding example above, where $C3$ is unfolded into the second antecedent of $C2$. After the unfolding, the TDAG structure of Fig. 4 will change to that shown in Fig. 5. As a result, the expected cost `$Root` of the root node will also change. In both cases the statistics of $C1$ play an identical role, so we can ignore this clause in the calculations. Before unfolding, the expected cost of clause $C2$ is $\$C2$, a measured quantity. After unfolding, the expected cost of the $C2.1$ and $C2.2$ clauses is $\$C2.1 + (1 - p(C2.1))\$C2.2$. The values of the likelihood $p(C2.1)$ and the costs $\$C2.1$ and $\$C2.2$ are, of course, not known quantities, but they can be predicted approximately from available TDAG measurements.

Let us illustrate with the case of $p(C2.1)$. $(1-p(C2.1))$ is the probability that clause $C2.1$ fails. This can happen if the head of the clause does not unify with the `p(...)` goal, or if the antecedent (`tail-q`) fails. $C2.1$ will fail to unify with the goal if either the more general clause $C2$ will not unify or if $C2$ unifies only to have the subgoal $C3$ fail. Both these likelihoods can be estimated from the TDAG statistics: $p(C3)$ is a measured quantity; and if $C2$ was attempted, say, 100 times and $C7$ only 75 times, then we infer that 25 times out of 100 the clause $C2$ failed due to non-unification of the head with the subgoal, so the probability that $C2$ fails to unify is 0.25. Similarly we can infer from the TDAG statistics the likelihood that the first antecedent of $C2.1$ (`tail-q`) fails. This antecedent is stronger than the `tail-q` antecedent of $C2$ since the substitution `a = X` has been applied to it. In Fig. 4, if we attempted $C2$ 100 times, $C7$ 75 times, and $C3$ only 10 times, then out of 75 times, we infer that the first subgoal (`tail-q`) succeeded only 10 times; hence the failure probability is about 65/75. Multiplying this by the likelihood that $C3$ fails to unify gives us our estimate of the likelihood that the `tail-q` subgoal of $C2.1$ fails.

Details apart, the point is that the TDAG statistics have the data necessary to compute the expected cost of solving the goal after the unfolding and to predict whether the unfolding will be beneficial. If the estimated cost of `$Root` with the unfolded clauses is lower than that without the unfolding, the optimizer goes ahead with it. The number of clauses for the unfolded predicate (p in this example) will increase by one; the fallback case—clause $C2.2$ in our example—must be present to preserve the semantics in case the unfolded clause ($C2.1$) fails. When there are more than two clauses for a predicate, deciding where in the list of clauses to place the fallback case is problematic. My approach was to place it last in the list of clauses and let clause reordering in the next pass determine its best position.

6 The Learn-Optimize Cycle

We have seen how the learning element collects statistics from program executions and incrementally builds a TDAG that can predict the optimal clause orderings at various points of the search and find advantageous unfolding transformations. The dynamic optimization process for a program is an alternating cycle of learning and optimizing passes: learning from sample executions, then transforming the program, learning from sample executions of the transformed program, transforming again, and so forth. The cycle stops when the optimizer can recommend no further transformations.

Two policies govern the choice of transformations during each cycle. First, clause reordering has priority over unfolding transformations. If according to the

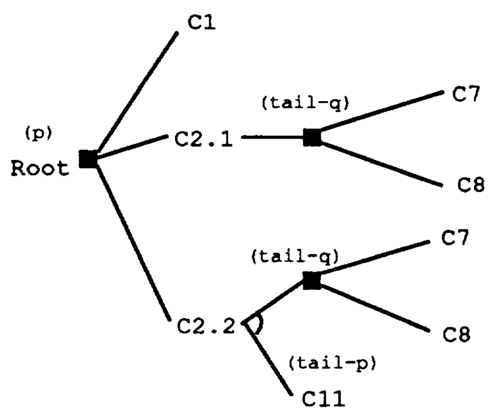

Figure 5: TDAG structure of Fig. 4 after unfolding clause $C2$.

TDAG the clauses for a particular subgoal are not in optimal order, and simultaneously one of the clauses is a candidate for unfolding, then the optimizer will perform *only* the clause reordering transformation. (An exception is the case where a clause has likelihood one in solving a subgoal; in this case, both the reordering and the unfolding can be performed.) The reason is that the reordering may affect the statistics used to evaluate the potential unfolding transformation, so the clauses should be in the right order before unfolding any of them.

Second, priority is given to transformations at nodes closest to the root. If a transformation is applied to a node on one pass of the cycle, no descendents of that node are transformed on the same pass. For example, if in Fig. 4 we reorder clauses $C1$ and $C2$, then any reordering of clauses $C7$ and $C8$ will have to wait until the next pass, even if the TDAG statistics currently recommend that $C8$ should be first.

The result of these two policies is that optimizations tend to occur deeper in the TDAG with each cycle, and thus the number of transformations performed tends to increase with each cycle. As noted above, several learn-optimize passes are used instead of one because a transformation at a TDAG node changes the statistics of the nodes below it and, as a result, the potential utility of any transformations at those deeper nodes.

Each transformation increases the number of clauses in the Prolog program, and over the entire cycle the program size may increase several fold. This is not a problem since program performance depends hardly at all on the size of the program. In the Prolog used for the experiments, clauses were retrieved from the database by a hash table indexed by the predicate functor. In some Prolog implementations, however, clauses are indexed by both the predicate functor and the leftmost functor of the first argument; in this case, the TDAG nodes would likewise be labeled by the pair of functor names, rather than by the predicate functor name alone.

Recall that the cycle terminates when the optimizer finds no justifiable transformations in the TDAG. If h is the maximum height of the TDAG tree, and if the sample size of the learning phase is large enough, then with high likelihood $3h$ is the maximum expected number of learn-optimize passes in the cycle. The factor of 3 arises from the possibility that, at any node C, a reordering of its child clauses may occur on one pass, an unfolding on the next, and a further reordering of the fallback clause from the unfolding on the one after that. On subsequent passes, transformations may occur at descendents of C but are not expected at C. There is also a statistical chance that the probability of success and the expected cost of attempting a clause may change a lot when the order of the clause is changed, so that a clause reordering may be reversed on the next pass. In my experiments, however, this did not occur; and in fact twelve passes were the most required for any program, compared to the theoretical limit of 21.

7 Experimental Results

To evaluate this dynamic optimization method, I modified a Prolog compiler so that the programs it compiles will construct their clause-name trees and collect the cost statistics as part of the search for a refutation of the input goal. (In the tests, I used both unification counts and CPU time as cost measures, with comparable results for the two measures.) After finding each solution to the input goal, the programs present the final clause-name tree to a multi-string TDAG learning element. More than ten Prolog programs of varying sizes were used to test the system. For each program, a problem generator was also constructed to provide a random set of problems for the (compiled) Prolog program to run. In most cases the generator was somewhat skewed so that, instead of problems being generated more or less uniformly by size, some regularities occured with higher probability. In some cases, the same program was run with different problem generators to assess the sensitivity of the optimizer to the problem source.

Next, the unoptimized program was put through a learn-optimize cycle. A sample size was empirically determined, as described above. This number was then used consistently by the learning element, although the actual set of problems changed on each pass. After compiling the program with the modified compiler and collecting statistics with the TDAG, useful transformations were identified and implemented. Then the optimized program was recompiled with the modified compiler, and the learn-optimize cycle continued.

The task of locating and installing the optimizing

transformations was done by hand with machine assistance; moreover, the transformations were selected and installed one at a time, rather than in batches. This procedure—which ordinarily would (and should) be entirely automatic— was adopted as a research tool to study in detail the performance of the optimizer and to verify whether each recommended transformation improved performance as predicted.

Clause reordering transformations are easy to identify and install, but the procedure for finding and constructing good unfolding transformations is slow and cumbersome. Even with machine assistance, testing every possible unfolding for its expected utility value took time, and I felt the need for a simpler rule that would suggest effective unfoldings more quickly.

I shall describe in detail the results for one program: the familiar `member` predicate defining membership in a list.

```
/* member(X, Y) <- X is in the list Y. */
   [CM1]:    member(X, X._).
   [CM2]:    member(X, Y.Z) := member(X,Z).
```

This problem demonstrates quite well that costs, not just probabilities, must be considered during the optimization; moreover, both unfoldings and clause reorderings played important roles in its optimization.

In this test the list `Y` was always a list of thirty different integers, and the first argument `X` matched exactly one of the integers in the list. The problem generator was constructed so that the target element `X` occurs exactly once in the list `Y`, at a position more or less uniformly distributed between fourth and thirtieth in the list. In a stream of such problems, it is clear the clause CM_2 will be applied with much greater frequency than CM_1. If, therefore, we chose clauses solely by probability, CM_2 would always be tried before CM_1, and the optimizer would reorder the clauses so that clause CM_2 precedes CM_1. In such a program the procedure would be first to go through the list to the end and then backtrack, testing the target against each element of the list in reverse order. By contrast, the TDAG determined that in most circumstances the expected $p/\$C$ value of `CM1` was about half that of `CM2` and declined to reorder these two clauses.

For this problem the sample size was 200 problem instances. Eight rounds of optimization were needed to produce the final program (given in the appendix) with eighteen clauses.

Examining the optimized program, we note that `member` was unfolded so that the search for the target element `X` begins with the *fourth* element of the list (clause `M1`). Clauses `M2` through `M4` will never be needed with this problem generator, but the optimizer can't know that and includes them for completeness.

The recursive predicate `tail-member` (`M5` and `M6`) remained unchanged throughout the cycle. Note, however, that `member` calls, not `tail-member`, but `d151` and other newly created predicates as a result of clause reorderings and unfoldings in the TDAG, before it finally calls `tail-member` in clause `M14`.

The three-clause predicate `d151` is a copy of `tail-member` that was produced by a reordering (placing `M16` first) and and an unfolding that skips over two more elements of the list before resuming the search. `d165a` and `d165b` are copies of `tail-member` and serve only to provide context for `d165c`, for which the usual clause ordering is reversed. This reordering—whose utility was well justified by a reduction in average runtime—was quite unexpected and apparently the result of an unintended statistical pattern in the problem generator.

The program size grew from two clauses to eighteen after three unfoldings and three reorderings. Five other clauses generated during the cycle ended up being unreferenced as a result of subsequent transformations and were therefore eliminated. Average costs for this generator were reduced by 18.5% for unifications and 17.2% for CPU time.

Note, finally, that all these changes are truly *dynamic* optimizations: the `member` source code alone will not point to these changes as effective. The examples—specifically, the statistical properties of the examples—are essential for understanding the utility value of these transformations to the original two-clause program.

Space will not permit describing all our other experiments, but let us mention a few highlights:

- The `implemented-by` program used by several researchers, e.g., (Shavlik, 1990; Subramanian and Feldman, 1990), is notorious for producing generalization-to-N anomalies in explanation-based generalization. In tests with three different problem generators, the optimizer produced only clause reorderings, never any unfoldings. Cost improvements ranged from about 4% percent for a generator with little skew to about 25% for one with strong skew.
- `color` is a brute-force graph-coloring algorithm using an unsophisticated backtracking search algorithm. Problems were generated more or less at random. No performance improvement was expected, and none was observed, despite several clause reorderings and one unfolding transformation. Significantly, however, no performance degradation was observed either.
- The best cost improvements—about 40%—occurred for a program that parses a context-free language. The gains resulted mainly from unfoldings and take advantage of strong patterns in the productions of the grammar.

Finally, the most striking feature of these experiments was the robustness of the results: several runs of the cycle with the same program and generator (but a different random seed for the generator) almost always resulted in the same sequence of optimizations. I had a strong sense that optimizer was finding a local minimum for the program's runtime performance, and that this minimum was not very sensitive to the particular sequence of the examples. This stands in contrast to reported results with EBL methods.

8 Summary: the Learn-Optimize Model

Although the details of this work pertain to dynamic optimization of Prolog programs, the major idea applies more generally: dynamic optimization should begin with a learning element that analyzes a sufficiently large sample for the learning to be reliable, followed by an optimization phase that bases its changes to the program on the results of the learning.

By contrast, most previous efforts to apply EBL to program speedup have been based upon a caching method: the idea is to save the solutions to individual problems and reuse them when the problems recur. The EBG algorithm generalizes the solution somewhat before caching it, but paradoxically the larger—and more informative—the solution, the weaker the generalization (Laird and Gamble, 1990a). This is the basis for the "generalization-to-N" problems. With caching comes the necessity for utility estimation: because of the limited space in the cache and limited time to search that space, only the most useful chunks can be retained. Caching also suffers from sensitivity to the order of arrival of the examples: because a decision must be made at once whether to save or discard a solution, unrepresentative problems that occur early will slow down the program until they can be displaced by more typical chunks at some much later time.

Our experiments exhibited no generalization-to-N problems and negligible sensitivity to the input order. No special "utility evaluation" has to be added on because utility analysis is the very core of the method. The learning element learns what it needs to evaluate the potential program transformations. In our case, since the only two transformations were clause reordering and unfolding, the statistics learned by the TDAG were those necessary to perform these transformations.

The optimization technique described here is not specific to Prolog. The same basic method is directly applicable to any nondeterministic typed-term language (Laird and Gamble, 1990b), including lambda-calculus-based and combinator-based languages. The relevant characteristic of these languages is that nondeterminism is represented explicitly in the language; the determinism necessary for consistent computation is provided by the underlying operational semantics (e.g., SLD resolution in Prolog). By contrast, imperative languages like C probably would not benefit much from dynamic optimization, since the nondeterministic element (search) is not represented as such in the code, but instead is embedded in *if–then–else* constructs or procedure calls.

Before adopting the learn-optimize cycle, I first tried the approach of modifying the Prolog interpreter to call the TDAG for search-control advice. Even when the TDAG *always* recommended the correct clause, the overhead of calling the learning element overwhelmed any cost savings, and I was never able to reduce the average CPU time below that of the unoptimized program running without the TDAG. Only then did I decide to use the TDAG, not for search control, but for guidance on program optimization. In effect, the method described in this paper compiles the search-control information into the program instead of calling for it at run time.

The procedure described in this paper is a first attempt at dynamic optimization for Prolog and suffers from a number of weaknesses that are being addressed by continuing research. The decision to pass the clause-name tree to the TDAG at the conclusion of a successful search means that only very limited failure statistics can be collected, and calls to higher-order predicates (like or and `call`) are not fully represented in the TDAG statistics. Also, by evaluating clause probabilities in the order in which the clauses occur in the program, the learning algorithm could recommend a transformation on one cycle and undo it on the next. (This never occurred in the experiments, however.) But the most unsatisfactory aspect of this (and related) research is the lack of any characterization of how "optimal" the resulting program will be. In the learn-optimize cycle the changes to the programs are based on a hill-climbing model: the program performance is expected to improve after each cycle, and optimization stops only when a local optimum is reached or TDAG size limits prevent further progress. There may be circumstances, however, where the truly *optimum* program can be reached only after transformations that temporarily worsen its performance; in such cases no hill-climbing method will find such an optimum.

In this paper we assume that the problem generator chooses problems independently from a distribution—i.e., the choice of the next input does not depend in any way upon previous inputs. One can imagine circumstances where this assumption does not hold, and yet our method provides no way to carry state information from one problem to the next. The TDAG can easily retain state information from one problem to the next; not so easy, however, is incorporating this state information into the optimized program.

9 Acknowledgments

My thanks to Wray Buntine, Oren Etzioni, Smadar Kedar, Steve Minton, Ron Saul, and the reviewers for their suggestions, and to Peter Norvig who provided a version of his elegant Prolog (Norvig, 1991). Part of this work was conducted during my stay at the Electrotechnical Laboratory in Tsukuba, Japan. My thanks to Dr. Taisuke Sato and the other members of the Machine Inference Section at that laboratory. This research was partially supported by the National Science Foundation (INT-9008726).

References

Abe, N. and Warmuth, M. 1990. On the computational complexity of approximating distributions by probabilistic automata. In *Proc. 3rd Workshop on Computational Learning Theory.*

Bell, T. C.; Cleary, J. G.; and Witten, I. H. 1990. *Text Compression.* Prentice Hall, Englewood Cliffs, N.J.

Etzioni, O. 1990. Why PRODIGY/EBL works. In *Proceedings, AAAI-90.* American Association for Artificial Intelligence. 915–922.

Gooley, M. and Wah, B. 1989. Efficient reordering of Prolog programs. *IEEE Trans. on Knowledge and Data Engineering* 1:470–482.

Gratch, J. and Dejong, G. 1991. A hybrid approach to guaranteed effective control strategies. In *Proc. 8th International Workshop on Machine Learning.* 509–513.

Greiner, R. and Orponen, P. 1991. Probably approximately optimal derivation strategies. In *Proceedings 2nd International Conference, Knowledge Representation and Reasoning.* 277–288.

Greiner, R. 1989. Towards a formal analysis of EBL. In *Proc. Sixth Int. Machine Learning Workshop.* Morgan Kaufmann. 450–453.

Laird, P. and Gamble, E. 1990a. EBG and term-rewriting systems. In *Proceedings, First International Workshop on Algorithmic Learning Theory,* Tokyo, Japan.

Laird, P. and Gamble, E. 1990b. Extending EBG to term-rewriting systems. In *Proceedings AAAI-90.* American Association for Artificial Intelligence.

Laird, P. 1988. Efficient unsupervised learning. In Haussler, D. and Pitt, L., editors 1988, *Proceedings, 1st Comput. Learning Theory Workshop.* Morgan Kaufmann.

Laird, P. 1991. Explanation-based generalization: theory meets experiment. Technical Report FIA-91-01-10-7, NASA Ames Research Center, AI Research Branch.

Laird, P. 1992. Discrete sequence prediction and its applications. In *Proc., 9th National Conference on Artificial Intelligence.* AAAI.

Letovsky, S. 1990. Operationality criteria for recursive predicates. In *Proceedings, AAAI-90.* American Association for Artificial Intelligence. 936–941.

Markovitch, S. and Scott, P. D. 1989. Utilization filtering: a method for reducing the inherent harmfulness of deductively learned knowledge. In *Eleventh IJCAI.* IJCAI. 738–743.

Minton, S. 1989. *Learning effective search control knowledge: an explanation-based approach.* Kluwer Academic Press.

Norvig, P. 1991. *Paradigms of A.I. Programming: Case Studies in Common LISP.* Morgan Kaufmann.

Prieditis, A. and Mostow, J. 1987. Prolearn: Towards a Prolog interpreter that learns. In *Proceedings of AAAI-87.* Morgan Kauffman.

Segre, A.; Elkan, C.; Scharstein, D.; Gordon, G.; and Russell, A. 1992. Adaptive inference. In Chipman, S. and Meyrowitz, A., editors 1992, *Machine Learning: Induction, Analogy, and Discovery.* Kluwer Academic.

Shavlik, J. 1990. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning* 5:39–70.

Smith, D. E. and Genesereth, M. R. 1985. Ordering conjunctive queries. *Ordering conjunctive queries* 26.

Sterling, L. and Shapiro, E. 1986. *The Art of Prolog.* M.I.T. Press.

Subramanian, D. and Feldman, R. 1990. The utility of EBL in recursive domains. In *Proceedings, AAAI-90.* American Association for Artificial Intelligence. 942–949.

Tamaki, H. and Sato, T. 1984. Unfold/fold transformation of logic programs. In *2nd International Logic Programming Conf.*

Yamada, S. and Tsuji, S. 1990. Computing the utility of EBL in a logic programming environment. In *Proceedings, JSAI National Meeting.* Japanese Society for Artificial Intelligence. 103–106. (In Japanese).

Yamada, S. 1992. Computing the utility of EBL in a logic programming environment. *J. of Japanese Society for Artificial Intelligence* 7(2):309–319. (In Japanese).

Appendix: Optimized member program

```
[M1]:  member(Item, [X1,X2,X3 | Rest]) :-
                      d151(Item, Rest).
[M2]:  member(Item, [Item . Rest]).
[M3]:  member(Item, [X . Rest]) :-
                      d173(Item, Rest).
[M4]:  member(Item, [X1,X2 | Rest)) :-
                      d173(Item, Rest).
[M5]:  tail-member(Item, [Item | Rest]).
[M6]:  tail-member(Item, [X | Rest]) :-
                      tail-member(Item, Rest).
[M7]:  d173(Item, [Item | Rest])).
[M8]:  d165a(Item, [Item | Rest]).
[M9]:  d165a(Item, [X | Rest]) :-
                      d165b(Item, Rest).
[M10]: d165b(Item, [Item | rest]).
[M11]: d165b(Item, (X | Rest)) :-
                      d165c(Item, Rest).
[M12]: d165c(Item, [X | Rest]) :-
                      d276(Item, Rest).
[M13]: d165c(Item, [Item | Rest]).
[M14]: d276(Item, [X | Rest]) :-
                      tail-member(Item, Rest).
[M15]: d276(Item, [Item | Rest]).
[M16]: d151(Item, [Item | Rest]).
[M17]: d151(Item, [X1, X2 | Rest]) :-
                      d165a(Item, Rest).
[M18]: d151(Item, [X | Rest]) :-
                      d173(Item, Rest).
```